NavP Versus SPMD : Two Views of Distributed Computation

Lei Pan, Lubomir F. Bic, Michael B. Dillencourt, and Ming Kin Lai
School of Information & Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
{pan,bic,dillenco,mingl}@ics.uci.edu

ABSTRACT
We introduce a new view of distributed computation, called the NavP view, under which a distributed program is composed of multiple sequential self-migrating threads called DSCs. In contrast with those in the conventional SPMD style, programs developed in the NavP view exhibit the nice properties of algorithmic integrity and parallel program composition orthogonality, which make them clean and easy to develop and maintain. The NavP programs are also scalable. We use example code and performance data to demonstrate the advantages of using the NavP view for general purpose distributed parallel programming.

KEY WORDS
navigational programming, distributed sequential computing, distributed parallel computing, single program multiple data, message passing, views

1 Introduction

The overall goal of our research is a methodology that helps to develop easy and scalable distributed parallel programs from sequential or parallel algorithms. We incorporate *computation mobility*, which distinguishes our approach from others. **Computation mobility** is defined as the ability for the locus of computation to migrate in a distributed environment and continue as it meets the required data. It is facilitated by self-migrating threads. The programming of self-migrating threads is called **Navigational Programming** (**NavP**).

Distributed sequential computing (**DSC**), defined as computing with distributed data using a single locus of computation, plays a key role in our approach. In the form of a self-migrating thread, a DSC program increases the performance of its non-distributed sequential counterpart on large problems by eliminating disk paging at a cost of efficient network communication, meanwhile preserves *algorithmic integrity* and hence good programmability [1]. Moreover, DSC is not just about sequential programming, it also serves *distributed parallel computing* (**DPC**). If parallel programming starts from a sequential algorithm, our methodology would consist of two transformations: the first converts the sequential algorithm into a DSC program, and the second turns the DSC program into a DPC program. If the starting point is a parallel algorithm, we construct multiple concurrent DSC threads to cover all the computations. In both cases, computation mobility helps to achieve efficient and correct data access. And it also provides a new way of describing computations in a distributed environment.

A distributed computation can be represented in a two-dimensional space consisting of a spatial and a temporal dimension. In this 2D space, the computation can be further seen in two different *views*. A **view** represents how we envision or describe a phenomenon. One of the possible views is SPMD, or "single program, multiple data." SPMD is a classical programming style that is commonly used in writing message-passing distributed programs [2]. In SPMD, a programmer writes one piece of code which is executed by the processes on all the nodes. All processes work with the same program text, but different processes carry out different sections of the text. This is done as follows. In systems such as MPI, a program can call some system-provided functions to get the ID number of the caller and the total number of processes, respectively. Now the functionality of the processes can be coded in terms of their ID numbers using **if** and **else if** statements.

In this paper, we introduce a new view called the *NavP view*, and compare it with the classical SPMD view in the context of general purpose distributed programming. In the **NavP view**, the description of a computation follows the migration of its locus. Computation mobility enables the NavP view, and the NavP view provides a new perspective to developing distributed parallel programs.

The paper is organized as follows. Section 2 introduces a 2D representation of distributed computation. Section 3 uses one simple example to show how the two different views are used in distributed programming, and some advantages of using the NavP view. Section 4 presents a real-world example, parallel Cholesky factorization, and compares its two implementations using the two views. The last section contains some final remark.

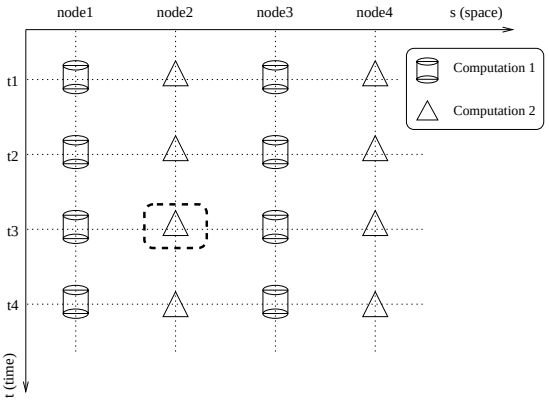

Figure 1. A two-dimensional representation.

2 A Two-dimensional Representation

We use $\mathbb{N}$ to denote the set that contains all the participating nodes in a distributed memory environment, and $\mathbb{T}$ to represent the set of time periods during which computations happen on these nodes. The Cartesian product of $\mathbb{N}$ and $\mathbb{T}$ (i.e., $\mathbb{N} \times \mathbb{T}$) forms a multiple-dimensional space. The set $\mathbb{N}$, which represents the spatial dimension, can be Cartesian product of sub-spaces, but for simplicity without losing generality we assume that spatially we have only one dimension throughout this paper. As depicted in Fig. 1, along the spatial dimension each discrete coordinate value represents one participating node (i.e., $\mathbb{N}$ = {node1, node2, node3, node4}), and along the temporal dimension each discrete coordinate value stands for a period of time (i.e., $\mathbb{T}$ = {t1, t2, t3, t4}). The shapes each represent a certain computation that happens in a specific location at a specific time. For example, the triangle in dashed-line box at (node2, t3) means that on node2, computation 2 is performed at time t3.

There are several issues that must be addressed when developing distributed programs. A dependency graph must be developed from a sequential or parallel algorithm. Data and computations must be mapped to the grid points in the 2D space. This mapping cannot violate the dependency relationship represented by the dependency graph, and should ideally satisfy two conflicting goals of maximizing parallelism and minimizing communication. These two steps produce a 2D representation of the computation. This 2D representation must then be turned into one or more sequential threads. Collectively these threads must cover every node and every edge of the dependency graph respecting the edge directions. The role of a view in fulfilling these tasks is to provide a way to group various computations into different threads.

3 Two Views of Distributed Computation

We describe two views, namely SPMD and NavP, with the help of a simple example.

3.1 Simple example

The distributed environment for the simple example is one that consists of two machines connected to each other on a network. In other words, our 2D world has two coordinate values, namely node1 and node2, along the spatial dimension. The example, with the pseudocode shown in Fig. 2(a), is a sequential program of three lines. Line (1) extracts the diagonal entries of matrix A and assigns them to vector $v1$, and lines (2) and (3) multiply the matrices B and A by two different vectors $v1$ and $v2$, respectively. In a non-distributed environment, the code would be executed as shown in Fig. 3(a). A dependency graph, as shown in Fig. 3(b), can be drawn based on the observation that computations at lines (2) and (3) depend on the intermediate results produced at lines (1) and (2), respectively. We further assume that A and B are large $N \times N$ matrices, and $v1$, $v2$, and $v3$ are vectors of size N, and A and B are distributed such that A is on node1 and B is on node2 respectively. The computations are then distributed to the 2D world as shown in Fig. 3(c) respecting the dependency relationship. To reduce communication, lines (1) and (3) are executed on node1 because the matrix A they use resides on node1, and line (2) is run on node2 since matrix B is on node2. The rule we use here for computation distribution is referred to as the principle of **pivot-computes**, which is defined as the principle under which a computation takes place on the node that owns the large-sized data. This node is called the **pivot node**. The principle suggests that the assignments of computations to nodes are done in such a way that large-sized data pieces (e.g., the matrices A and B) are not communicated across the network.

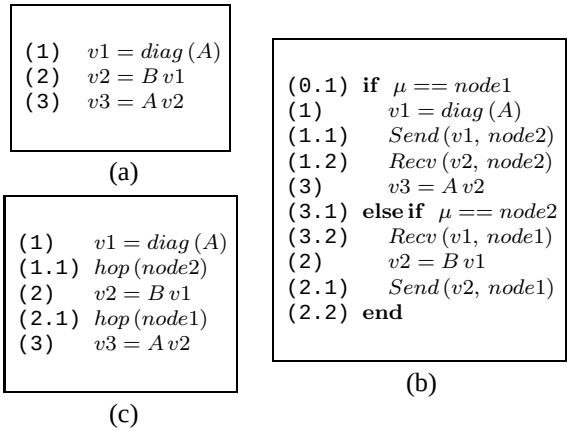

Figure 2. Computing on distributed data. (a) Sequential. (b) SPMD view. (c) NavP view.

3.2 The SPMD view

A view represents how we describe a phenomenon in a space. In the SPMD view, computations are described at

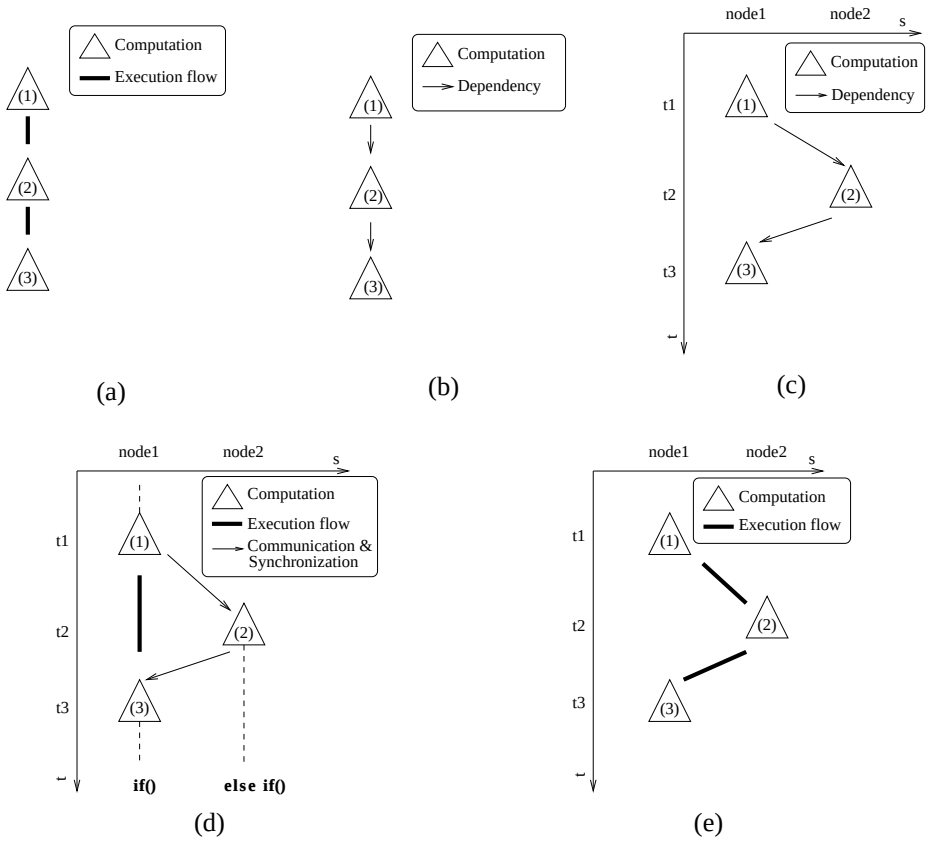

Figure 3. A sequential computation over distributed data. (a) Sequential. (b) Distribution. (c) Dependency graph. (d) SPMD view. (e) NavP view.

the spatial locations of the 2D distributed environment. The computations on each node are grouped into a thread which is run on that node. SPMD code is partitioned into blocks associated with different nodes, or the spatial coordinates, using **if** and **else if** statements, and within a block for each node, code lines are listed in the order in which they are executed along the temporal dimension.

The code in the SPMD view for the simple example is listed in Fig. 2(b), and its execution is visualized in Fig. 3(d). We make the following two observations: **1.** The original sequential code is restructured in the SPMD implementation. In particular, lines (1) and (3) are grouped together in the **if** block, because they are executed on node1. And line (2) is in the **else if** block, because it is executed on node2. Line (2) is no longer in between lines (1) and (3) in Fig. 2(b), as it used to be in the original code shown in Fig. 2(a); and **2.** Explicit communication and synchronization between nodes are needed even for the (distributed) sequential program. This is because there are multiple threads each running on a node despite the fact that the computation we are describing is sequential. The arrows in Fig. 3(d) depict the message flows that are used for communication and synchronization.

3.3 The NavP view

In the NavP view, the simple example is implemented using a self-migrating thread. We first put the relatively small data pieces $v1$ and $v2$ in agent variables (the ones that are carried by an agent), and the large-sized data pieces A and B in node variables (the ones that are stationary to a node) on node1 and node2, respectively. We then insert $hop()$ statements into the code, as shown in Fig. 2(c), in order for the computation locus to migrate to the right node where small and large sized data meet for the execution to happen. This implementation performs distributed sequential computing (DSC). The execution flow of the DSC is depicted in Fig. 3(e). The implementation uses the NavP view, in which the description of a computation follows the movement of its locus. Under the NavP view, sequential computations (distributed or not) are grouped into one thread, and this thread is self-migrating.

The simple example reveals two advantages of using the NavP view for distributed sequential computing.

1. The implementation shown in Fig. 2(c) preserves the original order of code lines. This property of DSC preserving the original code structure is referred to

as *algorithmic integrity* [1]. DSC also preserves loop structures, which can be seen from our case study presented in section 4.

2. No explicit communication and synchronization are needed in the NavP view, as can be seen if we compare Fig. 3(e) with Fig. 3(d). Under the NavP view, the sequential computations are grouped into one thread, in contrast to in the SPMD view multiple threads each running on a node. As such, communication and synchronization are intra-agent, and therefore are subsumed in the program execution flow. Across nodes, the locus of computation flows using $hop()$ statements. A DSC is seen as a 1D problem in the NavP view, rather than a 2D problem in the SPMD view. This simplification, however, does not mean that a NavP programmer has less control over communication or synchronization than a SPMD program does.

4 Case Study

In this section, we present parallel Cholesky factorization and its implementations in the two views. Cholesky factorization is an algorithm for factorizing symmetric positive definite matrices [3]. A positive definite matrix A can be factored into the product of two matrices $A = G\,G^T$, where G is a lower triangular matrix called the **Cholesky triangle**. This decomposition can then be used for different purposes, such as solving a linear system of equations of $Ax = b$. The Cholesky factorization algorithm takes A as its input and produces the matrix G. It works in place on A; when it concludes, the entries on and below the diagonal are the entries of G. For simplicity we will assume here that A is a dense matrix of $n \times n$ size.

```
(1) for k = 1 : n
(2)   A(k : n, k) /= sqrt(A(k, k))
(3)   updating (k, n)
(4) end
(5) updating (int k, int n)
(6)   for j = k + 1 : n
(7)     A(j : n, j) -= A(j : n, k)A(j, k)
(8)   end
(9) end
```

Figure 4. Pseudocode for sequential Cholesky factorization.

The sequential algorithm is listed in Fig. 4. There are two types of computations performed on the columns of the matrix A: **1.** *Scaling*: A column is scaled using its diagonal term (line (2) in Fig. 4). In each iteration of k, one column is scaled with the time complexity of $\Theta(n)$. The columns that have been scaled are called **G columns**. These columns will no longer be modified but will be used in later computation. Scaling processes all columns sequentially from left to right ($k = 1 : n$), i.e., a column is ready to be scaled only after all the columns to its left have been scaled and therefore turned into G columns, and after itself is updated using the information from all these G columns; **2.** *Updating*: A column is updated (line (7) in Fig. 4) using the values in all the G columns to its left. The work of updating for each iteration of k is $\Theta(n^2)$.

We distribute the data (i.e., the matrix A) to the participating nodes. The columns of matrix A are distributed to the nodes in a round robin fashion, in order to achieve better load balancing [3]. If the the final objective is a DSC program, a block fashion should be chosen. In order to transform the sequential algorithm into DSC, we insert two $hop()$ statements to the sequential code, one in each loop. The first $hop()$, inserted between lines (1) and (2) in Fig. 4, migrates the computation to the node that hosts the column being scaled, whereas the second $hop()$, inserted between lines (6) and (7) in Fig. 4, drives the computation to all the nodes in sequence carrying the G column from the latest scaling to perform updating. Before the second $hop()$, one assignment is used to load the most recent G column into an agent variable. Also, the loop indices k and j are stored in agent variables. The execution of the DSC program is visualized in Fig. 5(a), in which the number of nodes p is assumed to be 3. Each thick line represents a hop, some of which might be hopping to a node itself.

Now we transform the DSC program into a DPC program. The updatings on different nodes are independent of each other; but rather they all depend on the previous scaling, as depicted by the arrows in Fig. 5(a). Therefore, concurrent updating DSCs can be employed after each scaling step, as shown in Fig. 5(b).

The DPC program is listed in Fig. 6(b). There are two types of composing DSC programs: a single scaling DSC named $Scaler$ (with code lines (1)–(10)), and multiple updating DSCs named $Updaters$ (with code lines (11)–(18)). $Scaler$ carries the loop index k, an agent variable, that loops through all columns of matrix A. On the k^{th} iteration, $Scaler$ scales column k (line (4)). The function $col(k)$ maps the global column index k to a local column index; this function is needed because each node stores only a portion of the entire global matrix A. After scaling the column, $Scaler$ injects p $Updaters$ (lines (7.1)–(8.1)), and then it hops to the node that owns the next column of A (line (9.1)). The ID of this node is found using a column-to-node map function $node\_map()$. $Scaler$ then waits for the next round of computation. Each of the p $Updaters$ loads the newly computed G column k (again the local column index is $col(k)$) into its agent variables (line (12)), and then hops to the appropriate node (line (12.1)). In parallel, these p threads update the A columns for which they are responsible on all p nodes, using the G column stored in their agent variables and the matrix entries of A (line (15)). Two maps are used in the DPC code (lines (4), (9.1), (12), (12.1), and (15)). In particular, here the column-to-node

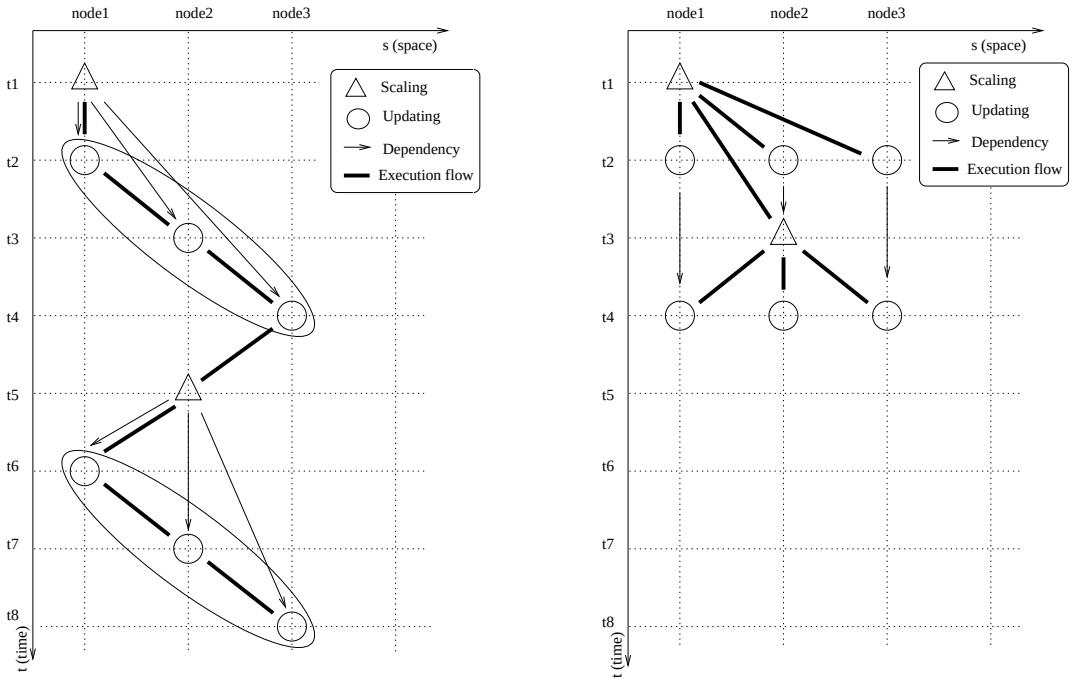

Figure 5. Cholesky factorization. (a) DSC. (b) DPC employing concurrent DSCs.

map is $node\_map(k) = (k-1)\%p + 1$, and the global-to-local-column-index map is $col(k) = (k-\mu)/p + 1$, where k is global column index, p is number of nodes, and μ is current node ID. The two types of DSCs, namely scaling and updating, interact with each other in the DPC program using only local injections or events. There are three lines of code that use $signalEvent()$ and $waitEvent()$ primitives (lines (9.2), (12.2), and (17.1)) in order to synchronize the threads running on the same node. $signalEvent()$ and $waitEvent()$ implement the classical operations of process blocking and wake-up. After $Scaler$ executes the $inject()$ command at line (8), it hops away immediately, and then the injected $Updaters$ start executing (line (12)). Thus $Scaler$ hops to the next node and continues its computation without having to wait for the injected $Updaters$ to hop away. $waitEvent()$ at line (9.2) makes $Scaler$ wait until the $Updater$ working on the same node finishes and signals an event Evt at line (17.1). $waitEvent()$ at line (12.2) makes sure that all the $Updaters$ from earlier iterations have finished before the current $Updater$ can start.

The MPI code adapted from reference [3] is listed in Fig. 7. It is obvious that the MPI code exhibits a significant departure from the original code structure. The two nested loops are both broken. The outer **for** loop over k is broken into smaller **while** loops over all the local columns that a node owns. Each process executing this code runs the **while** loop (line (2)) with loop index q. A global column index k, which is the same as the loop index k in Fig. 6(a) and (b), is being computed by all processes (lines (8) and (25)). The local column index q is mapped to its corresponding global position in the matrix A, and is then tested against the global index k (line (3)). If the test result at line (3) is true, the process owns the column that needs to be scaled. Therefore, it scales the column to get a new G column (line (4)), and passes the new G column to its right neighbor in the node ring (line (6)), before it uses the new G column to update the local A columns (line (11)). If the test result at line (3) is false, this process will receive the new G column from its left neighbor (line (15)), forward it to its right neighbor if needed (line (19)), and then update its local A columns (line (23)). The inner **for** loop over j in Fig. 4 is broken into two each appearing in an **if** or an **else if** code block. These smaller loops each update the local columns that a node owns. The access to the new G column is now through passed messages, rather than through shared variables in the sequential or NavP programs. Furthermore, as the **for** loop over j is broken down into smaller local loops, this data communication is relocated out of the loops, resulting in a fairly different and complicated termination condition (lines (16)-(18)).

Each distributed sequential computation (i.e., scaling and updating) is a 1D problem in the NavP view, and their DSC implementations preserve algorithmic integrity respectively. In particular, the nested loops are preserved in Fig. 6(b), as opposed to being broken in the SPMD view shown in Fig. 7. The code lines corresponding to different computation tasks are grouped in their own DSC programs as integral and unbreakable entities in the NavP view, re-

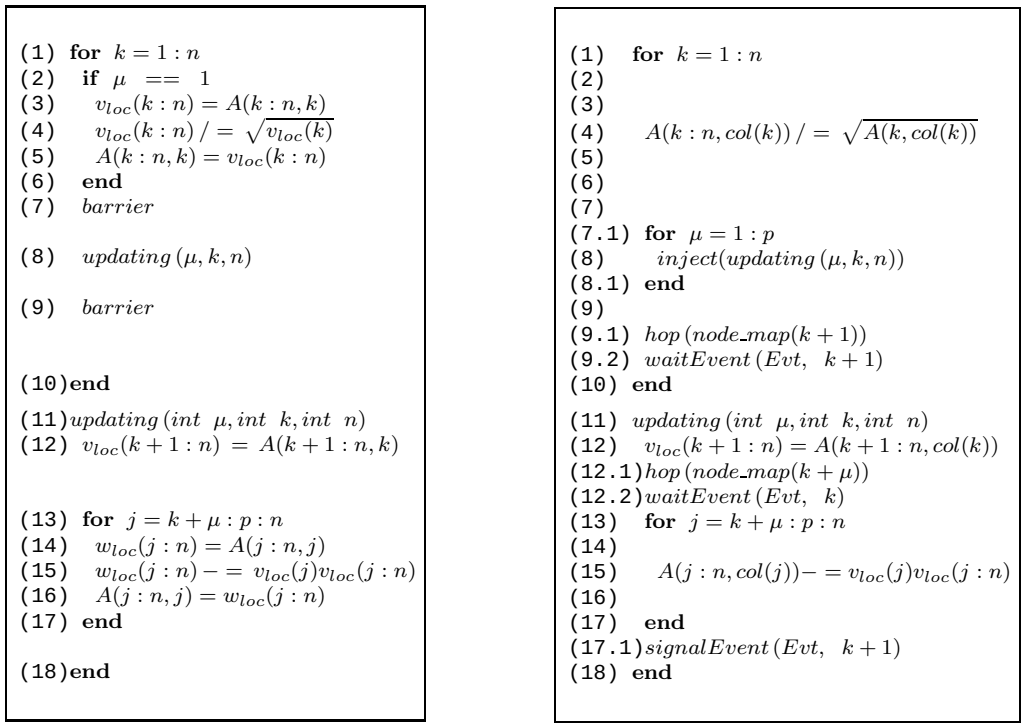

Figure 6. Pseudocode for parallel Cholesky factorization. (a) DSM. (b) DPC employing concurrent DSCs.

gardless of where they are executed. Multiple concurrent DSC programs are then composed into a DPC program. As can be seen from Fig. 6(b), different DSC programs only interact with each other with synchronization actions such as injections or events. These synchronization actions are all local to a node on which two or more threads that are being synchronized reside. The "intersection" between each composing DSC program and the resulting DPC program is minimum, and it consists of only local synchronizations. We say that each DSC is **orthogonal** to the DPC program. Parallel programming using self-migrating DSCs thus exhibits **composition orthogonality**. In contrast, in the SPMD view, code lines corresponding to different computation tasks are each broken into several pieces assigned to an **if** or **else if** block. Different code pieces corresponding to different computation tasks are then "tangled" together in an **if** or **else if** block in the temporal order in which they are executed on the node.

Distributed shared memory (DSM) can actually be used to alleviate the problem of code restructuring and tangling, even if the view being used is still SPMD. This is because on DSM code execution location does not have to be exact to guarantee the correctness of computation. Nevertheless, this may come with a high cost of communication, which makes it a non-scalable solution. For instance, in the simple example with pseudocode listed in Fig. 2, a DSM implementation would be the same as the sequential program, but the cost of its execution will be to ship the entire matrix A or B across the network. The principle of pivot-computes is violated. The DSM code for parallel Cholesky factorization, adapted from reference [3], is listed in Fig.6(a). This code is not as efficient as either our DPC or the MPI program. There are at least two reasons. First, the scaling part is not always done by the pivot node (i.e., the owner of the column being scaled). Rather, it is all done on the node with ID $\mu == 1$. This violates the principle of pivot-computes, and the result is that almost the entire matrix will be pulled to node1, which is more expensive than needed. Second, two $barriers$ are used for synchronization. A $barrier$ involves global communication and is in most cases a restriction that is stronger than necessary. In NavP, since all data accesses to variables are local, the only synchronization required is among different threads on the same node; no synchronization is necessary if two threads never meet each other. In other words, no inter-node synchronization is required. In the DPC program, the next round of scaling can start as soon as the local updating from the previous iteration is done, regardless of whether or not the remote updatings are finished, as depicted in Fig. 8(b). In contrast, the global $barrier$s (lines (7) and (9) in Fig. 6(a)) are less efficient. The next iteration can only start after all $Updaters$ from the previous iteration have finished, as depicted in Fig. 8(a). In a distributed environment with relatively high network latency and heterogeneous loads and processing powers, the performance improvement from this overlapping using local syn-

```
(1)k = 1; q = 1; col = μ : p : n; L = length(col)
(2)while q <= L
(3) if k == col(q)
(4)   A_loc(k : n, q) /= sqrt(A_loc(k, q))
(5)   if k < n
(6)     Send(A_loc(k : n, q), right)
(7)   end
(8)   k = k + 1
(9)   for j = q + 1 : L
(10)    r = col(j)
(11)    A_loc(r : n, j) -= A_loc(r, q)A_loc(r : n, q)
(12)  end
(13)  q = q + 1
(14) else
(15)  Recv(g_loc(k : n), left)
(16)  α = proc which sent k^th G col
(17)  β = index of right's final col
(18)  if right ≠ α and k < β
(19)    Send(g_loc(k : n), right)
(20)  end
(21)  for j = q : L
(22)    r = col(j)
(23)    A_loc(r : n, j) -= g_loc(r)g_loc(r : n)
(24)  end
(25)  k = k + 1
(26) end
(27)end
```

Figure 7. Pseudocode for parallel Cholesky factorization using MP in the SPMD view.

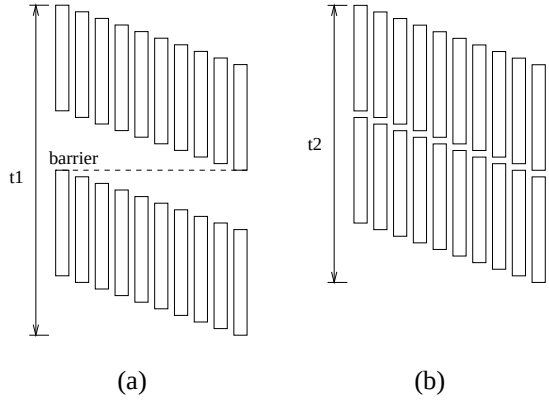

Figure 8. Synchronization. (a) Global with barriers. (b) Local in the NavP view.

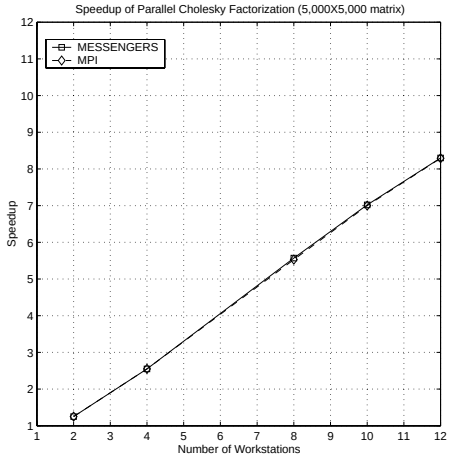

Figure 9. Performance of parallel Cholesky factorization.

chronizations can be significant. Similar to DSM, NavP allows the programmers to do shared variable programming [4], but in NavP programs the less efficient $barriers$ are not used for synchronization.

We implement the pseudocodes shown in Fig. 6(b) and Fig. 7 using MESSENGERS and MPI (LAM 6.5.9), respectively. The performance data is obtained from SUN Ultra Ultra 60's with 256MB of main memory, 1GB of virtual memory, and 100Mbps of Ethernet connection. These workstations have a shared file system (NFS). Fig. 9 shows the speedup data obtained from running both our DPC and the MPI programs. The speedup of our DPC program is almost the same as that of the MPI program, and their trends as the number of machines increases are the same which indicates same scalability. In order to make the NavP view practically useful, the $hop()$ statements in our code must be very efficient. This is addressed in our earlier work [5], and the key idea is to avoid moving code as the locus of computation migrates.

5 Final Remarks

In fluid dynamics [6], the **Eulerian** view considers changes as they occur at a fixed position in the fluid, while the **Lagrangian** view considers changes which occur as we follow a fluid particle along its trajectory. The Eulerian view is the SPMD view, and the Lagrangian view is our NavP view, if we replace the fluid space by the distributed environment, and the fluid particles by the self-migrating threads. Computations are assigned to the threads, and their changes are described following the migration of the "fluid particles."

Another analogy comes from our daily life. A train system provides two views. The two views are in the form of two types of train schedules. One type is the arrival and departure information shown on the monitors in all the train stations. This corresponds to the SPMD view because the descriptions are at fixed spatial locations. The other type of schedule is the brochures showing train numbers, their passing stations and the times. This is like the NavP view since the descriptions follow the trains' traces.

Fig. 10 shows a 2D system in which each directed curve, labeled as T1-T4, can be taken as a trace of a fluid particle, a train, or a self-migrating thread. Table 1(a) describes the system in the SPMD view. Each column corresponds to a spatial location. In contrast, each column in Table 1(b) describes the movement of one trace, which makes the table a NavP representation of the same system.

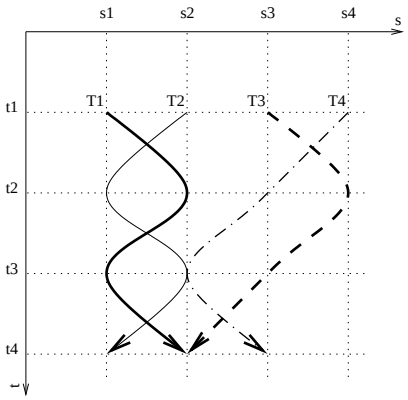

Figure 10. A two-dimensional system consists of four traces (T1-T4).

Table 1. The two views of the same system. (a) SPMD. (b) NavP.

	s1	s2	s3	s4
t1	T1	T2	T3	T4
t2	T2	T1	T4	T3
t3	T1	T2,T4	T3	
t4	T2	T1,T3	T4	

(a)

	T1	T2	T3	T4
t1	s1	s2	s3	s4
t2	s2	s1	s4	s3
t3	s1	s2	s3	s2
t4	s2	s1	s2	s3

(b)

The information provided by one view can be presented in another view. In distributed programming, this means that any programming task that a programmer can do in one view can be done in another view. Despite this, a particular view is good for a particular task. For example, the information of arrivals and departures shown on the monitors in one station is good for people that are in that station providing local services. This observation suggests that SPMD is good for client-server type of applications.

Composition orthogonality means that in a DPC program when new DSCs are added in, or existing DSCs are removed, the behaviors of other DSCs remain mostly intact, except for adding or removing some synchronizations. This avoids dramatic change in code structure. In contrast, an SPMD program describes all tasks that will run on a node in an **if** or **else if** block; this corresponds to a "vertical cut" at a spatial location in Fig. 10. The code for different tasks are thus tangled in the blocks. The SPMD view makes the job of adding or removing a task difficult.

It may not be possible to see in real life all the spatial locations such as the train stations collapse into one. But in distributed programming, this corresponds to the backward uniprocessor compatibility of a distributed program. Programs developed in the NavP view are backward uniprocessor compatible, because after the $hop()$ statements are ignored, they become multi-threaded programs on a uniprocessor machine. The DSCs coordinate with each other using events and injections, and they each preserve their sequential code structures. Such a multi-threaded program is a special case of a distributed parallel program in the NavP view. The same is not true for the SPMD view, since the location related information is used not only in the $Send()$ and $Recv()$ statements, but also in code restructuring (i.e., **if** and **else if** statements). An SPMD program looks strange on a uniprocessor machine, and the already changed code structure cannot be easily restored to the original shape. It is awkward to see messages sent to and from the same node itself. Maintaining such code for uniprocessor machines is unnecessarily difficult.

The use of the NavP view is only possible in distributed programming with the introduction of computation mobility carried by mobile agents. Strong mobility pioneered by the mobile agent community is the cornerstone of the new vision. In return, the usefulness of the new view is likely to encourage further research in mobile agents [7].

References

[1] L. Pan, L. F. Bic, and M. B. Dillencourt, "Distributed sequential computing using mobile code: moving computation to data," in *Proceedings of the 2001 International Conference on Parallel Processing (ICPP 2001)*, L. M. Ni and M. Valero, Eds. Los Alamitos, Calif.: IEEE Computer Society, Sept. 2001, pp. 77–84.

[2] C. Leopold, *Parallel and Distributed Computing: A Survey of Models, Paradigms, and Approaches*. New York: John Wiley & Sons, 2001.

[3] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 3rd ed. Baltimore, Md.: Johns Hopkins University Press, 1996.

[4] L. Pan, L. F. Bic, and M. B. Dillencourt, "Shared variable programming beyond shared memory: Bridging distributed memory with mobile agents," in *Proceedings of the 6th International Conference on Integrated Design & Process Technology (IDPT-2002)*, H. Ehrig, B. Kramer, and A. Ertas, Eds. Grandview, Texas: Society for Design & Process Science, June 2002.

[5] C. Wicke, L. F. Bic, M. B. Dillencourt, and M. Fukuda, "Automatic state capture of self-migrating computations in MESSENGERS," in *Proceedings, Second International Conference on Mobile Agents, MA '98*, ser. Lecture Notes in Computer Science, K. Rothermel and F. Hohl, Eds., vol. 1477. Berlin, Germany: Springer-Verlag, Sept. 1998, pp. 68–79.

[6] D. J. Tritton, *Physical fluid dynamics*, 2nd ed. New York: Oxford University Press, 1988.

[7] D. Kotz, R. Gray, and D. Rus, "Future directions for mobile agent research," *IEEE Distributed Systems Online*, vol. 3, no. 8, 2002.